The Journal of Supercomputing (2023) 79:9635–9665
https://doi.org/10.1007/s11227-022-05017-x

UVaFTLE: Lagrangian finite time Lyapunov exponent extraction for fluid dynamic applications

Rocío Carratalá-Sáez[1] · Yuri Torres[1] · José Sierra-Pallares[2] · Sergio López-Huguet[3] · Diego R. Llanos[1]

Accepted: 16 December 2022 / Published online: 19 January 2023

Abstract

The determination of Lagrangian Coherent Structures (LCS) is becoming very important in several disciplines, including cardiovascular engineering, aerodynamics, and geophysical fluid dynamics. From the computational point of view, the extraction of LCS consists of two main steps: The flowmap computation and the resolution of Finite Time Lyapunov Exponents (FTLE). In this work, we focus on the design, implementation, and parallelization of the FTLE resolution. We offer an in-depth analysis of this procedure, as well as an open source C implementation (UVaFTLE) parallelized using OpenMP directives to attain a fair parallel efficiency in shared-memory environments. We have also implemented CUDA kernels that allow UVaFTLE to leverage as many NVIDIA GPU devices as desired in order to reach the best parallel efficiency. For the sake of reproducibility and in order to contribute to open science, our code is publicly available through GitHub. Moreover, we also provide Docker containers to ease its usage.

Keywords Finite time Lyapunov exponent · Lagrangian coherent structures · OpenMP · GPU · Multithreading · Multi-GPU

1 Introduction

Transport in dynamic systems is often studied in terms of particle trajectories in phase space. When applied to fluids, this approach is often referred to as *Lagrangian*. In the absence of molecular diffusion, passive tracers follow fluid particle trajectories that are solutions of

Rocío Carratalá-Sáez and Yuri Torres have contributed equally to this work.

✉ Rocío Carratalá-Sáez
rocio@infor.uva.es

Extended author information available on the last page of the article

$$\dot{\vec{x}} = \vec{v}(\vec{x}, t),$$

where the right-hand side is the velocity field of the fluid.

The practical interest in the solution of the above system of equations lies in the calculation of the so called *Lagrangian Coherent Structures (LCS)*: The most repelling, attracting, and shearing material surfaces that form the skeletons of Lagrangian particle dynamics [15].

Lagrangian Coherent Structures (LCS) are distinguished surfaces of trajectories in a dynamic system that exert a major influence on nearby trajectories over a time interval of interest. Physical phenomena governed by LCS include floating debris, oil spills, surface drifters and chlorophyll patterns in the ocean; or clouds of volcanic ash and spores in the atmosphere and coherent crowd patterns formed by humans and animals [15].

The determination of LCS is becoming very important in several disciplines, including cardiovascular engineering [21], aerodynamics [6] and geophysical fluid dynamics [29]. In all these disciplines, LCS helps the understanding of the local flow phenomena, since they can be broadly interpreted as *transport barriers* in the flow. A paradigmatic example could be the prediction of the drift of an oil spill in the ocean [30], since LCS predict zones with intense changes beforehand, which allows for early emergency and mitigation planning.

From the computational point of view, the extraction of LCS consists of two main steps: The flowmap computation and the resolution of Finite Time Lyapunov Exponents (FTLE). We have already explored the flowmap computation in a previous work [7], so in this work we focus on the computation of the FTLE.

As explained in detail in later sections, the FTLE computation consists of a series of linear algebra operations applied to each particle of the flow independently of the other particles. Thus, the FTLE computation is one of those computing-intensive problems that are divided into many independent tasks which can be executed in parallel without requiring any communication among them. They are called embarrassingly-parallel problems [22]. Many real problems are included in this category, such as index processing in web searches [2], bag-of-tasks applications [33], traffic simulations [4] or Bitcoin mining [3].

The parallelization of embarrassingly parallel problems might not require a very complex parallel design to take advantage of parallel computing environments; however, the high amount of computational work requires high performance computing (HPC) approaches.

High performance coprocessors, such as Graphics Processing Units (GPUs), have been widely adopted in modern supercomputing systems as hardware accelerators. Designed to exploit the inherent parallelism of an application, these throughput-oriented processors have been significantly adopted for General Purpose Computing, as reflected in the configuration of many supercomputers ranked in the higher positions of the TOP500 ranking [31].

The exploitation of such systems offers a higher peak performance and a better efficiency compared to the classical homogeneous cluster systems [5, 34]. Due to these advantages, and since the cost of building heterogeneous systems is low, they are being incorporated into many different computational environments, from academic research clusters to supercomputing centers. Particularly for the case of embarrassingly parallel algorithms, this type of coprocessors are ideal to improve performance, because of the high number of cores they provide.

For this work, we have implemented the FTLE computation and provided it with the ability to be executed in multithreaded environments (taking advantage of OpenMP), and also in systems equipped with NVIDIA GPU devices (using the CUDA parallel programming model).

In this work, we offer the following contributions:

- We provide an in-depth description of the FTLE computation process, detailing the different operations on which it relies.
- We leverage the use of OpenMP and CUDA programming models to accelerate the FTLE computation, thus improving the performance of the LCS extraction procedure.
- We conduct a comprehensive performance evaluation that aims to evaluate the scalability of our solution when executed on multicore architectures, as well as when taking advantage of manycore GPU devices. Our evaluation includes the use of different combinations of architectures and different GPU devices, as well as OpenMP scheduling policies, in order to analyze their relative performance in the FTLE computation, for both 2D and 3D flows. Moreover, we also evaluate the performance when using up to four GPUs.
- We have released all the code developed in this work, both as source code[1] and as a set of Docker containers[2], allowing a quick, painless way to enable reproducibility and contributing to open science.
- For the sake of reproducibility and the contribution to open science, we have also published a Python code[3] that can be used to generate the mesh (either in 2D or 3D) and associated flowmap values that are the input of our FTLE computation code. Note that this code can also take advantage of multithreaded environments, thanks to the Python modules `multiprocess` and `joblib`.

The rest of the paper is structured as follows. In Sect. 2 we describe the mathematical background of the FTLE computation; in Sect. 3 we summarize the most important contributions to the FTLE computation that already exist; in Sect. 4 we illustrate how we have implemented the FTLE computation, as well as the input data and the applied parallelism strategies; in Sect. 5 we showcase the performance attained (using different platforms, OpenMP scheduling policies, and GPU devices); in Sect. 6 we set out the main conclusions of this work; lastly, in Sect. 7 we give some insights regarding our future work.

[1] Available at: https://github.com/uva-trasgo/UVaFTLE.

[2] Available at: https://hub.docker.com/r/rociocarratalasaez/uvaftle (multiple tags for different source images: devel (contains source code) and runtime (only contains the compiled binaries) images of nvidia/cuda)

[3] Available at: https://hub.docker.com/r/rociocarratalasaez/uvaftle-mesh-generation.

2 Finite time Lyapunov exponent

The Finite Time Lyapunov Exponent (FTLE) is defined as

$$\Lambda_{t_0}^{t_1}(\vec{x}_0) = \frac{1}{t_1 - t_0} \log \sqrt{\lambda_n(\vec{x}_0)},$$

where λ_n is the maximum eigenvalue of the Cauchy–Green strain tensor C, defined as follows

$$C(\vec{x}_0) = \left[\nabla F_{t_0}^{t_1}(\vec{x}_0)\right]^T \nabla F_{t_0}^{t_1}(\vec{x}_0)$$

and F is the flowmap [6].

The FTLE is a scalar field that works as an objective diagnostic for LCS: A first-order approach to assess the stability of material surfaces in the flow under study, by detecting material surfaces along which infinitesimal deformation is larger or smaller than off these surfaces [15].

Although more reliable mathematical methods have been developed for the explicit identification of LCSs, the FTLE remains the most used metric in the field for LCS identification.

3 Related work

In this section, we reference some existing works that offer optimizations in the context of the FTLE computation. Some develop and incorporate optimization techniques to efficiently exploit multicore systems; others take advantage of the computational capacity of GPU devices through CUDA or OpenGL programming models.

Some works, such as those presented in Sadlo et al. [27], Nouanesengsy et al. [23], Kuhn et al. [18], Chen and Shen [8], and Wang et al. [32], speed up the calculations of the FTLE problem by applying some optimization techniques such as reducing I/O, optimizing the use of the memory hierarchy, or using multiple CPUs. However, these works do not use hardware coprocessors such as GPUs, FPGAs or the Intel Neuromorphic microchip. These many core systems contain thousands of single cores that could be beneficial for these types of largely parallel problems.

Garth et al. [13], Dauch et al. [12] and Hlawatsch et al. [16] leverage GPU devices to process particle advection using the FTLE computation. In these works, there is no discussion regarding the details of their implementation, nor an in-depth description of the GPU optimization, since they do not focus on the computational viewpoint. In the work from Lin et al. [19], there is a wider description of the algorithm, including details regarding how to leverage the GPU computational power for the FTLE computation, but the source code is not available. Moreover, all these works rely on a single GPU device, and a multi-GPU scheme is not supported.

Conti et al. [10] propose the use of an Accelerated Processing Unit (APU) to fasten the computation of FTLEs for bluff body flows. However, the extraction of

Lagrangian Coherent Structures is not their objective. In their proposal, the mixed use of CPU and GPU, physically integrated in this kind of devices, avoids the communications costs between CPUs and GPUs. The communication latency in this kind of devices is reduced by the absence of a PCIe; nevertheless, the integrated GPU usually has less computational power than the one offered by dedicated ones. On top of that the authors use OpenCL 1.0, which was launched ten years ago and is not capable of fully exploiting hardware accelerators such as current NVIDIA GPUs.

Garth et al. [14] present the GPUFLIC tool, which uses the FTLE for interactive and dense visualization of unsteady flows. GPUFLIC leverages the usage of GPUs to accelerate the computations of the FTLE and depict the flow animation through OpenGL model. However, this tool does not take advantage of recent parallel programming models such as CUDA, since it uses OpenGL and the version of the software used is deprecated; consequently, it is not possible to efficiently exploit the underlying GPU hardware details, and it would not be possible to use nowadays devices, such as NVIDIA, AMD or Intel GPUs.

Sagristà et al. [28] use the FTLE to understand advection in time-dependent flow. It provides an interactive analysis of trajectories and introduces the concept of FTLE aggregation fields. An old PyCuda [17] version (from ten years ago) is used to manage the CUDA kernels on NVIDIA devices. Although it could be updated, PyCuda would present limitations in terms of performance compared to CUDA. Moreover, the tool proposed in that work does not support more than one CPU plus a single GPU device concurrently.

To the best of our knowledge, in the existing literature, there is a lack of in-depth analysis of the FTLE computation from the point of view of the computational effort and algorithms that we aim to cover in this work. Moreover, the existing software is either based on old programming models or is not capable of exploiting modern GPU devices for resolving FTLE computations. Additionally, the existing tools to not take advantage of multi-GPU heterogeneous platforms for resolving the same FTLE problem. In contrast, in our proposal we leverage modern CUDA software to tackle NVIDIA devices, and we also offer support to take advantage of as many GPU devices as desired.

4 Our implementation

In this section we describe the data provided to our algorithm as input, the output generated, the main procedures of our algorithm, and the CUDA kernels we have defined to substitute part of those procedures when performing the computations in the NVIDIA GPU devices.

4.1 Data

The computation of the FTLE takes three data sets as input (stored in their corresponding files): (1) the coordinates of the mesh points; (2) the mesh faces defined by the mesh points; and (3) the flowmap defining the mesh point trajectories for a fixed time instant *t*. Moreover, that time instant *t* is also provided as input data, together with the scenario dimension (either two-dimensional or three-dimensional), and the number of vertices per simplex in the mesh (namely three vertices in the 2D case, and four in the 3D case, forming triangles and tetrahedrons, respectively).

From that input data, in our implementation, we generate the data structures and variables (of the specified data type) detailed in Table 1.

The output provided by our implementation is an array composed of the FTLE, formed by values of type `double`, and its dimension being `nPoints`.

As part of the preprocessing of the data, we implemented two procedures called `create_nFacesPerPoint_vector` (see Algorithm 1) and `create_facesPerPoint_vector` (see Algorithm 2) in order to, respectively, generate the following additional arrays:

- `nFpP[]` (number of faces per point): array of dimension `nPoints` and elements of type `int`. It contains, for each point, the number of faces of which that point is a vertex added to that same information for the previous vertices. In other words, if `nFpP[i]=n`, that means that the *i-th* point of the mesh belongs to `n-nFpP[i-1]` mesh faces if `i>0`, or to `n` mesh faces if `i==0`.
- `FpP[]` (indices of the faces per point): array of dimension `nPoints * nFpP[nPoints-1]` and elements of type `int`. It contains, for each point, the indices of the faces of which that point is a vertex. In other words, if the *i-th* mesh point belongs to `n` faces (according to the `nFpP` array information), then there will be `n` face indices stored in contiguous positions of the array `FpP` starting at `FpP[nFpP[i]-nFpP[nPoints-1]]` if `i>0`, or `FpP[nFpP[i]]]` if `i==0`.

Algorithm 1 create_nFacesPerPoint_vector

```
Require: $nPoints, nFaces, nVpF, faces[\,]$

for $ip$ in $range(nPoints)$ do
    $nFpP[ip] = 0;$
end for
for $iface$ in $range(nFaces)$ do
    for $ipf$ in $range(nVpF)$ do
        $ip = faces[iface \cdot nVpF + ipf]$
        $nFpP[ip] = nFpP[ip] + 1$
    end for
end for
for $ip$ in $range(nPoints)$ do
    $nFpP[ip] = nFpP[ip] + nFpP[ip - 1]$
end for
return $nFpP$
```

Table 1 Description of the variables and data structures generated in our implementation from the input files and parameters, including their name, data type, length, and description

	Type	Length	Description
`nDim`	`int`	1	Dimension of the space (2D or 3D)
`nPoints`	`int`	1	Number of points in the mesh
`coords[]`	`double`	`nPoints * nDim`	Array containing the coordinates of each of the mesh points
`nVpF`	`int`	1	Number of points defining each of the mesh simplex. In our case, the 2D meshes are partitioned into triangles and thus `nVpF`= 3; in 3D, the mesh is divided into tetrahedrons, so `nVpF`= 4
`nFaces`	`int`	1	Number of simplex (triangles/tetrahedrons) formed by the mesh points
`faces[]`	`int`	`nFaces * nVpF`	Array containing the indices of the mesh points that form each simplex vertices (those indices refer to the position of those vertices in the `coords` array)
`t_eval`	`double`	1	Time instant in which the flowmap has been calculated and FTLE will be computed
`flow[]`	`double`	`nPoints * nDim`	Array containing the flowmap values for the mesh points in the time instant `t_eval`

Algorithm 2 create_facesPerPoint_vector

```
Require: $nPoints, nFaces, nVpF, faces[\,], nFpP[\,]$

for $ip$ in $range(nPoints)$ do
    $count = 0$
    $iFacesP = (ip == 0)\ ?\ 0 : nFpP[ip - 1]$
    $nFacesP = (ip == 0)\ ?\ nFpP[ip] : nFpP[ip] - nFpP[ip - 1]$
    while $(iface < nFaces)\ \&\ (count < nFacesP)$ do
        for $ipf$ in $range(nVpF)$ do
            if $faces[iface \cdot nVpF + ipf] == ip$ then
                $FpP[ifacesP + count] = iface$
                $count = count + 1$
            end if
        end for
    end while
end for
return $FpP$
```

Note that, in all cases, when referring to "point index", that means the ordinal associated to that point or face starting from `0` and until `nPoints`, which means that the point coordinates will be stored starting in `coords[i*nDim]`. Equivalently, the "face index" is the ordinal associated to that point or face starting from `0` and until `nFaces`, which means that the indices of the face vertices (as many as `nVpF`) will be stored starting in `faces[i*nVpF]`.

4.2 Algorithm

The FTLE computation procedure, summarized in Algorithm 3, is formed by the following main steps:

1. Compute the gradients of the flowmap (see Algorithm 4 for 2D and Algorithm 5 for 3D).
2. Generate the tensors from the gradients and perform the matrix–matrix product of the previously generated tensors by their transposes (see Algorithm 6 for 2D and Algorithm 7 for 3D). Calculating the gradients is done based on the Green–Gauss theorem [20].
3. Compute the maximum eigenvector of each resulting matrix. Note that as we are computing the eigenvalues of matrices of size 2x2 (2D) or 3x3 (3D), which in practice means, respectively, solving a second and third degree equation, we have directly implemented this computation (see Algorithm 8 for 2D and Algorithm 9 for 3D), instead of calling mathematical libraries that perform this computation for generic matrices of any size.
4. Calculate the logarithm of the square matrix of the maximum eigenvalue and divide the result by the time instant to evaluate.

Algorithm 3 FTLE

Require: $nDim, t\_eval, coords\_file, faces\_file, flowmap\_file$

```
$nVpF = (nDim == 2)\ ?\ 3 : 4$                                    ▷ Triangles or tetrahedrons
$\{nPoints, coords\}$ = read_coordinates(coords_file)
$\{nFaces, faces\}$ = read_faces(faces_file)
$flow$ = read_flowmap(flowmap_file)
$nFpP$ = create_nFacesPerPoint_vector(nPoints, nFaces, nVpF, faces)
$FpP$ = create_FacesPerPoint_vector(nPoints, nFaces, nVpF, faces, nFpP)
for $ip\ in\ range(nPoints)$ do
    if $nDim == 2$ then
        $g1, g2$ = 2D_gradient (ip, nVpF, coords, flow, faces, nFpP, FpP)
        $max\_eigen$ = max_eigenvalue_2D([g1, g2])
    else
        $g1, g2, g3$ = 3D_gradient (ip, nVpF, coords, flow, faces, nFpP, FpP)
        $max\_eigen$ = max_eigenvalue_3D([g1, g2, g3])
    end if
    $result[ip] = log(sqrt(max\_eigen))/t\_eval$
end for
return $result[\,]$
```

Algorithm 4 2D_gradient

Require: $ip, nP, nVpF, coords[\,], flow, faces[\,], nFpP[\,], FpP[\,]$

```
$count = 0$
$nFaces = (ip == 0)\ ?\ nFpP[ip] : nFpP[ip] - nFpP[ip-1]$
for $iface\ in\ range(nFaces)$ do
    $idxface = (ip == 0)\ ?\ FpP[iface] : FpP[nFpP[ip-1] + iface]$
    for $ivert\ in\ range(nVpF)$ do
        $iv = faces[idxface \cdot nVpF + ivert]$
        if $(coords[iv \cdot nDim + 1] == coords[ip \cdot nDim + 1])$ then
            if $(coords[iv \cdot nDim] < coords[ip \cdot nDim])$ then
                $closeL = iv;\ count++;$                                  ▷ (i-1, j)
            end if
            if $(coords[iv \cdot nDim] > coords[ip \cdot nDim])$ then
                $closeR = iv;\ count++;$                                  ▷ (i+1, j)
            end if
        end if
        if $(coords[iv \cdot nDim] == coords[ip \cdot nDim])$ then
            if $(coords[iv \cdot nDim + 1] < coords[ip \cdot nDim + 1])$ then
                $closeD = iv;\ count++;$                                  ▷ (i, j-1)
            end if
            if $(coords[iv \cdot nDim + 1] > coords[ip \cdot nDim + 1])$ then
                $closeU = iv;\ count++;$                                  ▷ (i, j+1)
            end if
        end if
    end for
end for
if $count == 4$ then
    $dx = coords[closestR \cdot nDim] - coords[closestL \cdot nDim]$
    $dy = coords[closestU \cdot nDim + 1] - coords[closestD \cdot nDim + 1]$
    $gra1[0] = (flow[closeR \cdot nDim] - flow[closeL \cdot nDim])/dx$
    $gra1[1] = (flow[closeR \cdot nDim + 1] - flow[closeL \cdot nDim + 1])/dx$
    $gra2[0] = (flow[closeU \cdot nDim] - flow[closeD \cdot nDim])/dy$
    $gra2[1] = (flow[closeU \cdot nDim + 1] - flow[closeD \cdot nDim + 1])/dy$
end if
return $gra1, gra2$
```

Algorithm 5 3D_gradient

Require: $ip, nP, nVpF, coords, flow[\,], faces[\,], nFpP[\,], FpP[\,]$

```
$count = 0$
$nFaces = (ip == 0)\ ?\ nFpP[ip] : nFpP[ip] - nFpP[ip-1]$
for $iface\ in\ range(nFaces)$ do
    for $iv\ in\ range(nVpF)$ do
        if $(coords[iv \cdot nDim + 1] == coords[ip \cdot nDim + 1])$ then
            if $(coords[iv \cdot nDim] < coords[ip \cdot nDim])$ then
                $closeL = iv;\ count = count + 1;$                    ▷ (i-1, j, k)
            end if
            if $(coords[iv \cdot nDim] > coords[ip \cdot nDim])$ then
                $closeR = iv;\ count = count + 1;$                    ▷ (i+1, j, k)
            end if
        end if
        if $(coords[iv \cdot nDim] == coords[ip \cdot nDim])$ then
            if $(coords[iv \cdot nDim + 1] < coords[ip \cdot nDim + 1])$ then
                $closeD = iv;\ count = count + 1;$                    ▷ (i, j-1, k)
            end if
            if $(coords[iv \cdot nDim + 1] > coords[ip \cdot nDim + 1])$ then
                $closeU = iv;\ count = count + 1;$                    ▷ (i, j+1, k)
            end if
        end if
        if $(coords[iv \cdot nDim] == coords[ip \cdot nDim])$ then
            if $(coords[iv \cdot nDim + 1] < coords[ip \cdot nDim + 1])$ then
                $closeB = iv;\ count = count + 1;$                    ▷ (i, j, k-1)
            end if
            if $(coords[iv \cdot nDim + 1] > coords[ip \cdot nDim + 1])$ then
                $closeF = iv;\ count = count + 1;$                    ▷ (i, j, k+1)
            end if
        end if
    end for
end for
if $count == 6$ then
    $dx = coords[closestR \cdot nDim] - coords[closestL \cdot nDim]$
    $dy = coords[closestU \cdot nDim + 1] - coords[closestD \cdot nDim + 1]$
    $dz = coords[closestF \cdot nDim + 2] - coords[closestB \cdot nDim + 2]$
    $gra1[0] = (flow[closeR \cdot nDim] - flow[closeL \cdot nDim])/dx$
    $gra1[1] = (flow[closeU \cdot nDim] - flow[closeD \cdot nDim])/dx$
    $gra1[2] = (flow[closeF \cdot nDim] - flow[closeB \cdot nDim])/dx$
    $gra2[0] = (flow[closeR \cdot nDim + 1] - flow[closeL \cdot nDim + 1])/dy$
    $gra2[1] = (flow[closeU \cdot nDim + 1] - flow[closeD \cdot nDim + 1])/dy$
    $gra2[2] = (flow[closeF \cdot nDim + 1] - flow[closeB \cdot nDim + 1])/dy$
    $gra3[0] = (flow[closeR \cdot nDim + 2] - flow[closeL \cdot nDim + 2])/dz$
    $gra3[1] = (flow[closeU \cdot nDim + 2] - flow[closeD \cdot nDim + 2])/dz$
    $gra3[2] = (flow[closeF \cdot nDim + 2] - flow[closeB \cdot nDim + 2])/dz$
end if
return $gra1, gra2, gra3$
```

Algorithm 6 2D_tensor_product

Require: $gra1, gra2$

```
$ftle\_m[0] = gra1[0] \cdot gra1[0] + gra1[1] \cdot gra1[1]$
$ftle\_m[1] = gra1[0] \cdot gra2[0] + gra1[1] \cdot gra2[1]$
$ftle\_m[2] = gra2[0] \cdot gra1[0] + gra2[1] \cdot gra1[1]$
$ftle\_m[3] = gra2[0] \cdot gra2[0] + gra2[1] \cdot gra2[1]$
$gra1[0] = ftle\_m[0];\ gra1[1] = ftle\_m[1]$
$gra2[0] = ftle\_m[2];\ gra2[1] = ftle\_m[3]$
$ftle\_m[0] = gra1[0] \cdot gra1[0] + gra1[1] \cdot gra1[1]$
$ftle\_m[1] = gra1[0] \cdot gra2[0] + gra1[1] \cdot gra2[1]$
$ftle\_m[2] = gra2[0] \cdot gra1[0] + gra2[1] \cdot gra1[1]$
$ftle\_m[3] = gra2[0] \cdot gra2[0] + gra2[1] \cdot gra2[1]$
return $ftle\_m$
```

Algorithm 7 3D_tensor_product

```
Require: $gra1, gra2$
  $ftle\_m[0] = gra1[0] \cdot gra1[0] + gra2[0] \cdot gra2[0] + gra3[0] \cdot gra3[0]$
  $ftle\_m[1] = gra1[0] \cdot gra1[1] + gra2[0] \cdot gra2[1] + gra3[0] \cdot gra3[1]$
  $ftle\_m[2] = gra1[0] \cdot gra1[2] + gra2[0] \cdot gra2[2] + gra3[0] \cdot gra3[2]$
  $ftle\_m[3] = ftle\_m[ip \cdot nDim \cdot nDim + 1]$
  $ftle\_m[4] = gra1[1] \cdot gra1[1] + gra2[1] \cdot gra2[1] + gra3[1] \cdot gra3[1]$
  $ftle\_m[5] = gra1[1] \cdot gra1[2] + gra1[1] \cdot gra2[2] + gra3[1] \cdot gra3[2]$
  $ftle\_m[6] = ftle\_m[ip \cdot nDim \cdot nDim + 2]$
  $ftle\_m[7] = ftle\_m[ip \cdot nDim \cdot nDim + 5]$
  $ftle\_m[8] = gra1[2] \cdot gra1[2] + gra2[2] \cdot gra2[2] + gra3[2] \cdot gra3[2]$
  $gra1[0] = ftle\_m[0];\ gra1[1] = ftle\_m[1];\ gra1[2] = ftle\_m[2]$
  $gra2[0] = ftle\_m[3];\ gra2[1] = ftle\_m[4];\ gra2[2] = ftle\_m[5]$
  $gra3[0] = ftle\_m[6];\ gra3[1] = ftle\_m[7];\ gra3[2] = ftle\_m[8]$
  $ftle\_m[0] = gra1[0] \cdot gra1[0] + gra1[1] \cdot gra1[1] + gra1[2] \cdot gra1[2]$
  $ftle\_m[1] = gra1[0] \cdot gra2[0] + gra1[1] \cdot gra2[1] + gra1[2] \cdot gra2[2]$
  $ftle\_m[2] = gra1[0] \cdot gra3[0] + gra1[1] \cdot gra3[1] + gra1[2] \cdot gra3[2]$
  $ftle\_m[3] = gra2[0] \cdot gra1[0] + gra2[1] \cdot gra1[1] + gra2[2] \cdot gra1[2]$
  $ftle\_m[4] = gra2[0] \cdot gra2[0] + gra2[1] \cdot gra2[1] + gra2[2] \cdot gra2[2]$
  $ftle\_m[5] = gra2[0] \cdot gra3[0] + gra2[1] \cdot gra3[1] + gra2[2] \cdot gra3[2]$
  $ftle\_m[6] = gra3[0] \cdot gra1[0] + gra3[1] \cdot gra1[1] + gra3[2] \cdot gra1[2]$
  $ftle\_m[7] = gra3[0] \cdot gra2[0] + gra3[1] \cdot gra2[1] + gra3[2] \cdot gra2[2]$
  $ftle\_m[8] = gra3[0] \cdot gra3[0] + gra3[1] \cdot gra3[1] + gra3[2] \cdot gra3[2]$
  return $ftle\_m$
```

Algorithm 8 max_eigenvalue_2D

```
Require: $M$
  $sq \leftarrow sqrt(M[21]*M[21]+M[10]*M[10]-2*(M[10]*M[21])+4*(M[11]*M[20]))$
  $eig1 \leftarrow (M[21] + M[10] + sq)/2$
  $eig2 \leftarrow (M[21] + M[10] - sq)/2$
  return $(eig1 > eig2)\ ?\ eig1 : eig$
```

Algorithm 9 max_eigenvalue_3D

Require: M

```
$a \leftarrow -1$
$b \leftarrow M[10] + M[21] + M[32]$
$c \leftarrow M[12] * M[30] + M[22] * M[31] + M[11] * M[20 - M[10] * M[21] - M[10] * M[32] - M[21] * M[32]$
$d \leftarrow M[10] * M[21] * M[32] + M[11] * M[22] * M[30] + M[12] * M[20] * M[31] - M[10] * M[22] * M[31] - M[11] * M[20] * M[32] - M[12] * M[21] * M[30]$
                    ▷ Solve 3rd degree equation and return the maximum of the solutions
$A \leftarrow b * b - 3 * a * c$
$B \leftarrow b * c - 9 * a * d$
$C \leftarrow c * c - 3 * b * d$
$del \leftarrow B * B - 4 * A * C$
if $A = B \;\&\; A = 0$ then
    $max \leftarrow -b/(3 * a)$
else
    if $del = 0$ then
        $max \leftarrow (-b/a + B/A) > ((-B/A)/2)?(-b/a + B/A) : ((-B/A)/2)$
    else
        $T \leftarrow (2 * A * b - 3 * a * B)/(2 * A * sqrt(A))$
        $xt \leftarrow acos(T)/3$
        $x1 \leftarrow (-b - 2 * sqrt(A) * cos(xt))/(3 * a)$
        $x2 \leftarrow (-b + sqrt(A) * (cos(xt) + sqrt(3) * sin(xt)))/(3 * a)$
        $x3 \leftarrow (-b + sqrt(A) * (cos(xt) - sqrt(3) * sin(xt)))/(3 * a)$
        $max \leftarrow (x1 > x2)?x1 : x2$
        $max \leftarrow (max > x3)?max : x3$
    end if
end if
return $max$
```

4.3 Multithreaded CPU

In our implementation, we have used the OpenMP directive `#pragma omp parallel for` prior to the for-loop in the FTLE algorithm that iterates over the number of points in the mesh. That is, in Algorithm 3, we have inserted the mentioned directive after the sixth line (i.e., after the generation of the `FpP` array).

As we shall see in Sect. 5, we have explored the performance when equipping that directive with different scheduling policies:

- `static`: Equal number of iterations are assigned to each OpenMP thread.
- `dynamic`: The iterations are dynamically assigned to the threads as soon as they are finished with their previous work.
- `guided`: Similar to the last one, but the chunk size (number of iterations to assign) starts off large and decreases to better handle the load imbalance between iterations.

4.4 CUDA kernels and grid

The full algorithm presented in the previous section is GPU-enabled. In our implementation for CUDA kernels, we create as many threads as points exist in the mesh that is being evaluated. The threads of the kernel concurrently execute the computation derived from each mesh point (see Sect. 4.2). Thus, the first for-loop that iterates over the number of points in the mesh for the FTLE algorithm (see Algorithm 3) is parallelized.

Inside each GPU kernel, before starting the execution of the algorithm, two operations are performed. In the first operation, the global identifier of each thread is calculated. Each identifier corresponds to a mesh point. For the code simplicity, we use one-dimensional threadBlock and grid. Thus, the following instruction is executed to calculate the thread global identifier:

$$int\ \ th\_id = blockIdx.x * blockDim.x + threadIdx.x;$$

In the second operation, it is checked that the number of threads that are launched is not larger than points contained in the mesh. For that we insert the following condition wrapping the FTLE implementation:

$$if(th\_id < numCoords)\ \{...\}$$

Each thread of the GPU grid executes exactly the sequence of steps described in Sect. 4.2. The instructions are the same as those in the multithread implementation. The global memory access pattern is not perfectly coalesced, since contiguous mesh points are not stored in contiguous positions of the GPU global memory. This implies that contiguous threads access mesh points that are stored in non-contiguous memory addresses.

Tuning strategies to improve performance, such as coalescing, prefetching, unrolling and occupancy maximization, are introduced in the classical CUDA reference manuals, such as NVIDIA [24]. Additionally, the nvprof CUDA profiling tool [25] enables to better understand the hardware resources utilization and the performance of the applications. An intuitive idea is that the best option when choosing the threadBlock size is trying to maximize the multiprocessors occupancy to disguise latencies when accessing the global device memory.

According to the NVIDIA Reference Manual [24], the threadBlock sizes that maximize the GPU occupancy are 256, 512, and 1024. All these threadBlock sizes have been evaluated, and the best results have been obtained for a threadBlock size of 512 threads. Moreover, note that the default device behavior is the same as if we indicated to prioritize the L1 cache memory with “cudaFuncCachePreferL1” configuration, because we are not using shared memory. This cache configuration reduces the global memory transactions size and thus, the data traffic is decreased for not-perfectly coalesced memory access patterns, such as ours.

We maintain a one-dimensional threadBlock geometry, as it makes it easier to calculate the global index of each thread reducing the number of kernel instructions. We now describe the threadBlock sizes employed:

- 2D kernel: The recommended block sizes that maximize the GPU occupancy have been evaluated and the best results have been obtained for threadBlock of 512 threads and L1 cache memory with “cudaFuncCachePreferL1” configuration. This cache configuration reduces the global memory transactions size and thus, the data traffic is reduced for not-perfectly coalesced memory access patterns. The `nvprof` tool indicates that our 2D has a 100 % of hardware utilization (Stream-Multiprocessor (SM) occupancy), in spite of the large number of registers required.

- 3D kernel: This kernel requires much more registers than the 2D kernel. Based on the `nvprof` tool information, when using any of the recommended thread block sizes, the maximum occupancy we were able to reach was around 50%. For that reason, in this case we opted for an alternative block size, particularly of 668, that lets us maximize the number of active threads allocated in the SM. With this block size, the maximum occupancy is around 65% for this kernel. As we have already stated with respect to the 2D kernel, in this 3D kernel using the L1 cache memory with "cudaFuncCachePreferL1" configuration also offers the best results.

Loop unrolling has been shown to be a relatively inexpensive and beneficial optimization for GPU programs. Thus, we have unrolled the Algorithms 6 and 7 (2D and 3D, respectively).

Finally, we want to highlight that our software is currently capable of leveraging all the GPU devices available in a single node. Thus, we are performing our multi-GPU executions in a shared memory environment. We use the OpenMP programming model instantiating as many threads as GPU devices to distribute the load among them. Particularly, we have designed a static partitioning of the mesh points based on the number of GPU devices that take part of the execution.

5 Experiments

In this section, we first show a comparison of the execution time attained by our software, compared to that offered bu the LCSTool. After that we describe the platforms in which we have performed our software parallel performance experiments, and then we present the different evaluations we have conducted in them, as well as analyze the results obtained.

5.1 Comparison with existing software

Currently, there are three main open source projects that offer the same functionality as ours: VisIt [9], flowtk Package [1], and LCSTool [26] (already mentioned in Sect. 3). Unfortunately, VisIt and flowtk Package, although available, have no step-by-step documentation and are difficult to use. Nevertheless, the LCSTool software, which consists of a MATLAB code developed by the Haller group of ETH Zurich, is intuitive and usable through several MATLAB scripts that are already provided by their developers/maintainers. For this reason, in this section we present a comparison between UVaFTLE sequential execution time and the LCSTool one. Note that LCSTool has no parallel implementation, thus we cannot conduct any parallel performance comparison.

Table 2 Elapsed time (seconds) when computing five times the FTLE for a 2D case of 1,000K mesh points both for the LCSTool and the UVaFTLE software

Software	#1	#2	#3	#4	#5	Mean
LCSTool	8.990 s	8.660 s	8.690 s	8.640 s	8.780 s	**8.752 s**
UvaFTLE	0.105 s	0.105 s	0.109 s	0.105 s	0.110 s	**0.107 s**

The important numbers to bold are those in the "mean" column (as it has been done) in order to highlight them with respect to the others

We have chosen the Double-Gyre flow for the comparison, which is a simplification of a 2D double-gyre pattern that occurs frequently in geophysical flows [11]. As MATLAB is not available in any of the systems that we will use for evaluating the parallel performance of our software, we have conducted the tests to provide the comparison using a LENOVO laptop equipped with an Intel(R) Core(TM) i5-8250U CPU @ 1.60GHz (composed of 4 cores and a total of 8 threads). The OS is Ubuntu 20.04.4 LTS. For the LCSTool execution we have used MATLAB R2022b, and for the UVaFTLE execution we have used `gcc 9.4.0`.

Table 2 reflects the elapsed time (in seconds) associated to five different executions of the FTLE computation using the LCSTool and the UVaFTLE software, for a 2D mesh composed of 1,000K points. As it can be observed in the comparison, the UVaFTLE software is approximately 81 times faster than the LCSTool when computing the FTLE in this scenario.

5.2 Platforms used for the performance evaluation

The experiments presented in this paper were conducted in two different systems. The reason behind showcasing the performance in both of them is to illustrate the behavior of our implementation in two of the most popular architectures existing nowadays: AMD and Intel. The hardware features and operating systems (OS) of these two systems are the following:

- `Gorgon` computing server from the Universidad de Valladolid. This system features two AMD EPYC 7713 (Ryzen 3) CPU @ 2.0GHz, with 64 Core Processors and 128 threads each, and it is equipped with four NVIDIA GeForce GTX TITAN Black @ 3.417GHz, each provided with 5.9GB. The OS is Centos 7.
- `Finisterrae III` supercomputer from the Centro de Supercomputación de Galicia (CESGA). This system is composed of 354 nodes that feature two Intel Xeon Ice Lake 8352Y CPU @ 2.2GHz, with 32 Core Processors and 64 threads each, and two NVIDIA A100 GPU each @ 1.186 GHz and provided with 39.6GB. The OS is Rocky Linux 8.4.

In our experiments in `Gorgon`, we compile our implementation using `nvcc 11.3` with the flag `-arch=sm_35`, and including the flag `-march=znver3` to generate

Table 3 Description of the test cases used in our experiments

Dim	nPoints	nFaces	Min-max(x, y, z)	Length(x, y, z)
2D	1000K	1,996,002	(0–2, 0–1, 0–0)	(1000, 1000, 0)
2D	≈10,000K (9,998,244)	19,983,842	(0–2, 0–1, 0–0)	(3162, 3162, 0)
3D	≈ 500K (512,000)	2,958,234	(0–1, 0–1, 0–1)	(80, 80, 80)
3D	1000K	5,821,794	(0–1, 0–1, 0–1)	(100, 100, 100)

instructions that run on the third Generation of EPYC/RYZEN in an optimized manner. In our experiments in `Finisterrae III`, we compile our implementation using `nvcc 11.2`. In both systems, we incorporated the optimization options flag `-O3`, as well as the flag `-fopenmp` for the multithreaded and multi-GPU versions.

5.3 Experiments

We conducted several experiments in the mentioned platforms to test the scalability and efficiency of our implementation when taking advantage of CPU multithreading, and also leveraging up to four GPU devices.

To test the parallel performance, we opted for two test cases (one in 2D and the other in 3D) that arise from real-world scenarios. For the 2D case, we use the Double-Gyre flow already employed in the previous section. For the 3D case, we use the Arnold–Beltrami–Childress (ABC) flow or Gromeka–Arnold–Beltrami–Childress (GABC) flow, a 3D incompressible velocity field resulting from an exact solution of Euler's equation [35]. Table 3 summarizes the test cases dimensions, the number of mesh points and mesh simplex (either triangles or tetrahedrons), the interval of interest at each axis, and the number of elements in the interval at each axis taken to define the mesh points.

The speedup and efficiency shown in Tables 8 and 13 have been calculated according to their classical definition, as follows:

$$Speedup = \frac{t_{seq}}{t_{par}} \qquad Efficiency = \frac{Speedup}{n}$$

Being t_{seq} the sequential execution time, t_{par} the one associated to the parallel execution, and n either the number of CPU threads in the multithreaded parallel execution or the number of GPU devices in the multi-GPU case.

In all the experiments, we tested the performance attainable with our OpenMP-based implementation, covering three scheduling policies (static, guided, and dynamic), each of them using the default chunk size, as well as a GPU-based one. The experiments carried out in `Gorgon` used up to four GPU devices (NVIDIA Titan Black), and up to 128 threads; those conducted in `Finisterrae III` used up to two GPU devices (NVIDIA A100), and up to 64 threads.

Note that, in all cases, we have executed five times each experiment and we show the mean values.

Table 4 Execution time results (ms) for the 2D mesh that contains 1,000K points when using up to 128 CPU threads in `Gorgon` and different OpenMP scheduling policies

1000K	1th	8th	16th	24th	32th	40th	48th	56th	64th
Static	89.96	25.21	10	6.98	6.24	5.07	5.10	4.75	4.84
Dynamic	92.92	113.5	87.38	79.01	91.39	124.28	142	166.73	192.41
Guided	90.69	16.08	8.18	5.87	4.82	4.26	4.21	4.71	4.02

1000K	72th	80th	88th	96th	104th	112th	120th	128th
Static	4.56	4.71	5.61	5.5	5.68	5.97	7.46	15.97
Dynamic	218.88	240.76	264.44	290.45	313.23	340.67	381.57	422.36
Guided	4.4	4.84	4.9	5.33	5.48	7.68	11.1	13.44

Table 5 Execution time results (ms) for the 2D mesh that contains 10,000K points when using up to 128 CPU threads in `Gorgon` and different OpenMP scheduling policies

10,000K	1th	8th	16th	24th	32th	40th	48th	56th	64th
Static	904.46	152.21	86.06	53.86	51.36	39.65	40.14	34.66	37.05
Dynamic	932.91	1189.76	845.36	908.45	907.8	1141.88	1354.18	1681.66	1914.17
Guided	892.54	120.94	73.71	66.87	44.68	34.98	33.63	29.6	30.02

10,000K	72th	80th	88th	96th	104th	112th	120th	128th
Static	37.73	36.9	33.9	34.02	34.81	34.37	38.54	39
Dynamic	2354.66	2482.05	2740.34	2951.49	3254.92	3580.5	3862.96	4323.92
Guided	31.34	31.27	30.33	33.16	31.35	33.03	39.27	36.18

Table 6 Execution time results (ms) for the 2D mesh that contains 1,000K points when using up to 64 CPU threads in `Finisterrae III` and different OpenMP scheduling policies

1000K	1th	8th	16th	24th	32th	40th	48th	56th	64th
Static	91.12	12.6	7.01	5.3	4.64	4.51	4.88	5.25	5.54
Dynamic	119.91	66.27	63.99	64.45	70.75	76.03	79.34	83.42	91.1
Guided	90.48	12.16	7.5	5.56	5.18	5.23	5.32	6.05	8.06

Table 7 Execution time results (ms) for the 2D mesh that contains 10,000K points when using up to 64 CPU threads in `Finisterrae III` and different OpenMP scheduling policies

10,000K	1th	8th	16th	24th	32th	40th	48th	56th	64th
Static	908.06	116.5	60.21	42.77	32.96	29.72	27.01	30.68	32.89
Dynamic	1189.94	773.14	687.05	673.01	726.41	627.98	633.9	691	683.99
Guided	903.44	115.17	60.33	41.81	33.29	28.63	29.07	28.12	27.97

5.4 2D experiments

In this section, we first present the execution time attained by the OpenMP-based implementation for the two 2D meshes (of dimension 1000K and 10,000K points) in `Gorgon` and `Finisterrae III`. This can be seen in Tables 4, 5, 6 and 7.

Now, in Figs. 1 and 2, we compare the performance attained by the GPU-based implementation with that offered by the OpenMP one. Note that we have not represented the results associated to the `dynamic` policy for the sake of visibility (as it offers the worse results), nor the results associated to 1 thread for the same reason.

The main conclusions that derive from these experiments are:

- Regarding the different OpenMP scheduling policies, the `dynamic` one is the option that offers the worst results in all cases, while the `static` and `guided` ones have a similar behavior in general. This is due to the fact that all the FTLE computations performed have a very similar (and small) load for the different mesh points, and there is an enormous overhead caused by the management of the dynamic assignation of the iterations to the threads, compared to simply assigning collections of them, as is done with the `static` and `guided` policies.
- With respect to the results observed in `Gorgon`, in the small test case, the maximum speedup is reached with 72 threads for the OpenMP static scheduling option (19.73x) and with 64 threads for the guided one (22.55x). After that number of threads, the efficiency decreases due to the fact that the load per thread is not sufficiently large to be able to leverage the instantiated resources. In the big test case, the highest speedup is observed with 56 threads both for the static (26.10x) and the guided (30.16x) policies, but maintained longer at the same efficiency level when increasing the number of threads.

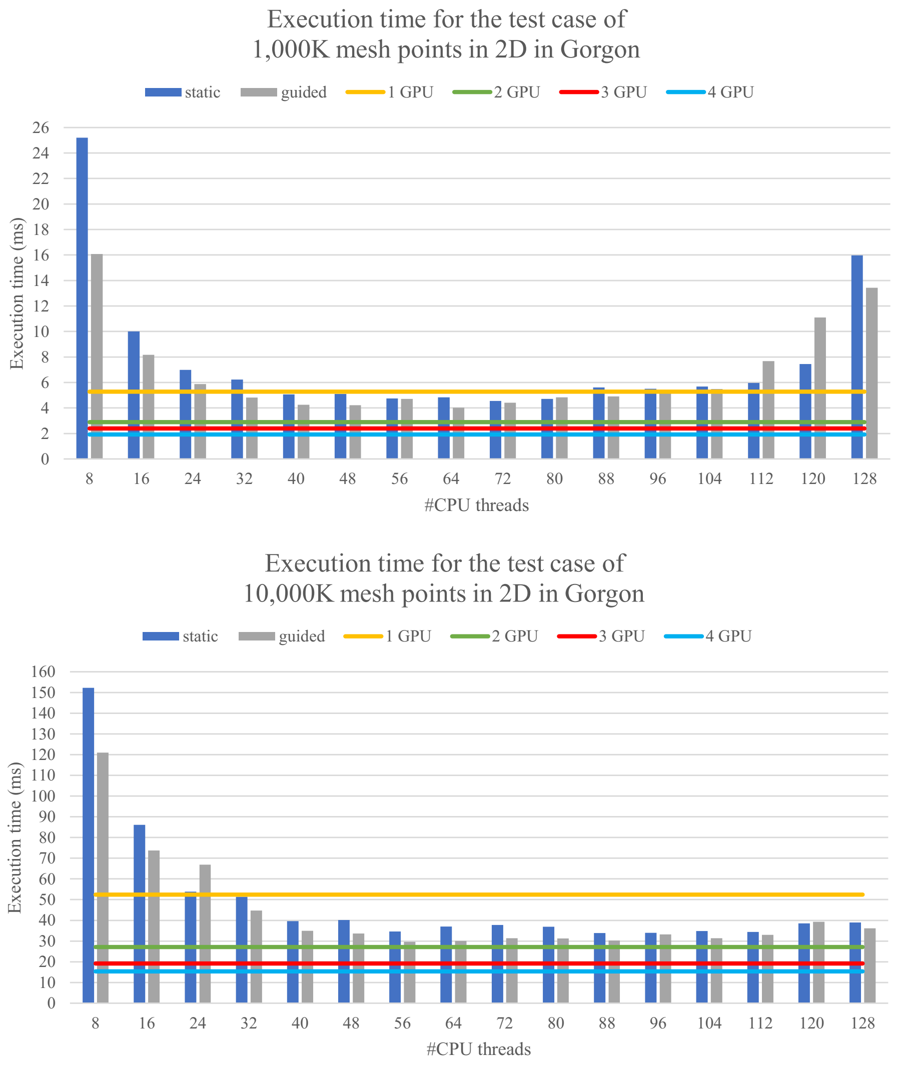

Fig. 1 Execution time for the 2D case with a mesh of 1000K points (top) and 10,000K points (bottom) in `Gorgon`. The GPU results are shown in horizontal lines, while the bars are used to represent the different execution time values associated to the CPU threads indicated in the X-axis

- Regarding the results observed in `Finisterrae III`, the best performance with the small test case is attained with 40 threads (20.19x speedup) with the static policy and 32 threads (17.47x speedup) with the guided one. With the big test case and the static policy the best speedup is observed with 48 threads (33.62x), and with 40 threads with the guided one (31.55x). Compared to what we observe in `Gorgon`, in this platform the performance is better maintained after reaching the best one because, in that case, we use fewer threads than in the other system.

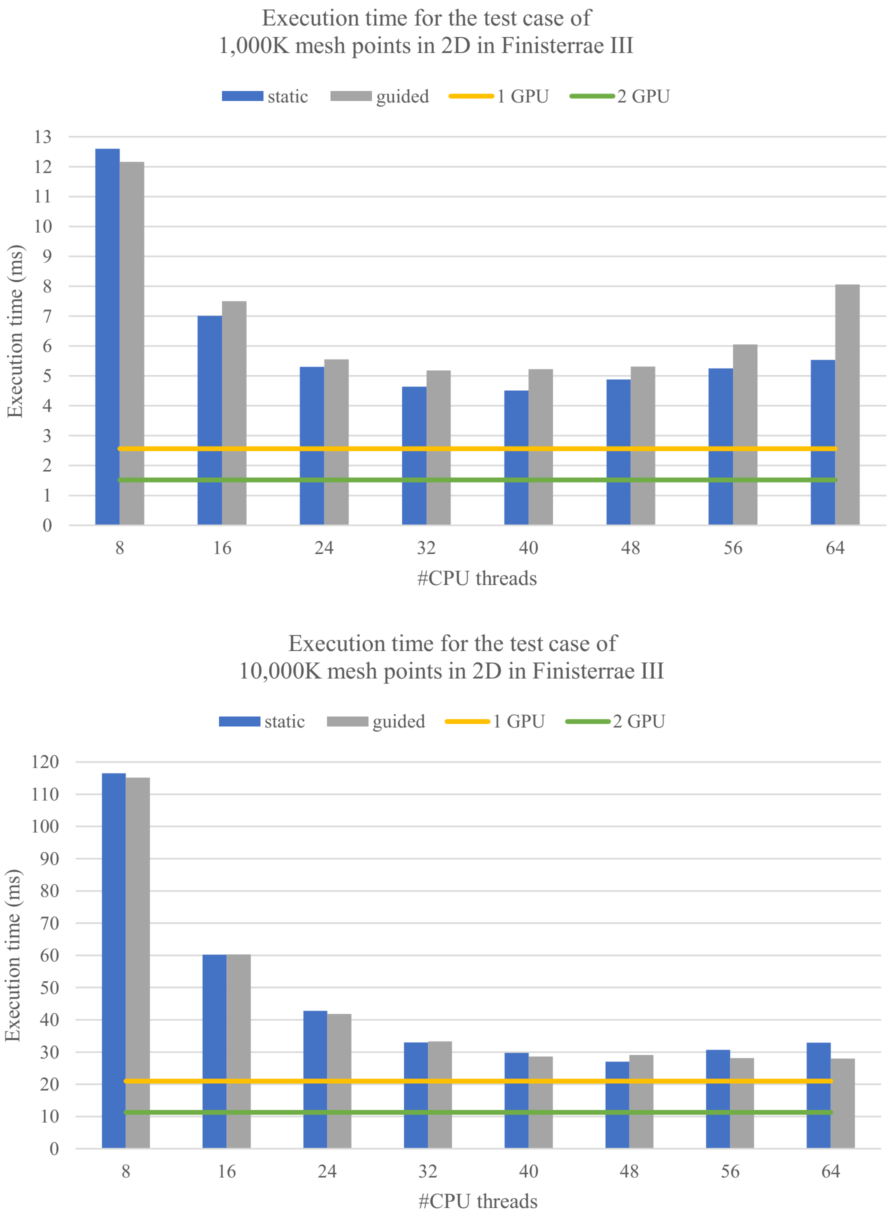

Fig. 2 Execution time for the 2D case with a mesh of 1000K points (top) and 10,000K points (bottom) in `Finisterrae III`. The GPU results are shown in horizontal lines, while the bars are used to represent the different execution time values associated to the CPU threads indicated in the X-axis

- Regarding the GPU-based performance, in `Gorgon` the comparison with the results observed when using CPU threads is clearly dependant on the dimensions of the mesh. When computing the FTLE for small meshes, we can see in

Table 8 GPU speedup and efficiency in `Gorgon` and `Finisterrae III` for the 2D small (left) and big (right) test cases

Platform	Small case-2D 1000K points				Big case-2D 10,000K points			
	Gorgon		Finisterrae III		Gorgon		Finisterrae III	
# GPU	Speedup	Efficiency	Speedup	Efficiency	Speedup	Efficiency	Speedup	Efficiency
2	1.8264	0.9132	1.6871	0.8435	1.9322	0.9661	1.8602	0.9301
3	2.2057	0.7352	–	–	2.7347	0.9116	–	–
4	2.7488	0.6872	–	–	3.4129	0.8532	–	–

the graphic that, from 40 to 80 CPU threads, the performance is better than that offered with only one GPU (and this also applies when using 32 or 88 threads with the `guided` policy). When the mesh is big, one GPU is never outperformed by any multithreaded CPU execution when using 40 or more threads with the static policy, or 32 or more with the guided one (due to a higher computational load). In any case, taking advantage of two or more GPU devices always outperforms the results observed with any multithreaded CPU execution. This is not applicable to `Finisterrae III`, where the NVIDIA device is much more powerful than the one in `Gorgon`, and thus even using a single GPU always outperforms any result obtained with CPU threads.

- Regarding the GPU scalability, as reflected in Table 8, we observe that the load in the small test case is not sufficient enough to fully take advantage of the multi-GPU computational power executions; thus, the efficiency is between 68 and 91% in `Gorgon`, and around 84% in `Finisterrae III`. When increasing the mesh dimension, we observe an improvement in the efficiency, it being always equal to or higher than 85%.

5.5 3D experiments

In this section, we present the execution time attained by the OpenMP-based implementation for the two 3D meshes (of dimension 500K and 1000K points) in `Gorgon` (Tables 9 and 10), as well as in `Finisterrae III` (Tables 11 and 12).

As we have shown for the 2D case, we also show in Figs. 3 and 4 a comparison of the performance attained by the GPU-based implementation and that offered by the OpenMP one, now based on the 3D-based experiments. Note that we have not represented the results associated to the `dynamic` policy for the sake of visibility (as it offers the worst results), nor the results associated to the lowest number of threads for the same reason.

The main conclusions that derive from these experiments are:

- Regarding the different OpenMP scheduling policies, we obtain the same conclusion as in the 2D case: the `dynamic` one is the option that offers the worst

Table 9 Execution time results (ms) for the 3D mesh that contains 500K points when using up to 128 CPU threads in `Gorgon` and different OpenMP scheduling policies

500K	1th	8th	16th	24th	32th	40th	48th	56th	64th
Static	169.37	46.34	21.93	15.27	12.69	12.24	13.04	12.32	12.56
Dynamic	173.33	117.01	117.26	212.19	279.29	339.57	403.78	476.28	422.71
Guided	170.24	26.26	19.58	16.46	15.29	16.52	16.79	17.57	18.95

500K	72th	80th	88th	96th	104th	112th	120th	128th
Static	13.09	12.59	13.1	13.65	15.18	14.31	20.35	21.88
Dynamic	438.2	470.22	487.64	504.84	529.17	555.99	576.93	593.29
Guided	20.08	20.03	24.35	24.76	26.22	28.23	32.61	35.15

Table 10 Execution time results (ms) for the 3D mesh that contains 1.000K points when using up to 128 CPU threads in `Gorgon` and different OpenMP scheduling policies

1000K	1th	8th	16th	24th	32th	40th	48th	56th	64th
Static	330.75	72.93	38.98	27.57	23.11	20.5	21.56	20.59	18.57
Dynamic	337.6	231.29	245.39	409.52	527.26	654.66	786.97	929.42	811.88
Guided	332.49	52	35.63	29.86	26.55	24.93	25.04	26.93	27.06

1000K	72th	80th	88th	96th	104th	112th	120th	128th
Static	20.98	21.08	21.28	22.15	22.21	22.32	28.47	32.41
Dynamic	867.39	914.43	971.14	1139.13	1038.84	1067.52	1134.77	1128.64
Guided	28.12	31.55	33.39	36.53	39.43	36.28	44.47	50.44

Table 11 Execution time results (ms) for the 3D mesh that contains 500K points when using up to 64 CPU threads in `Finisterrae III` and different OpenMP scheduling policies

500K	1th	8th	16th	24th	32th	40th	48th	56th	64th
Static	282.99	49.02	27.29	19.07	16.2	16.27	14.49	15.6	12.82
Dynamic	313.09	101.22	149.73	255.07	338.33	414.68	494.69	567.66	446.01
Guided	289.97	42.18	24.71	18.38	16.15	17.86	19.54	20.33	24.1

Table 12 Execution time results (ms) for the 3D mesh that contains 1,000K points when using up to 64 CPU threads in `Finisterrae III` and different OpenMP scheduling policies

1000K	1th	8th	16th	24th	32th	40th	48th	56th	64th
Static	529.52	90.59	50.68	35.42	28.04	25.9	25.66	22.93	25.97
Dynamic	589.66	196.42	302.96	491.6	643.47	812.54	946.05	1113.96	867.01
Guided	539.27	79.67	45.82	33.2	28.16	28.9	28.89	31.36	34.42

results in all cases. However, in this case, the differences between the `static` and `guided` policies are much more noticeable, due to the fact that now there is more load and the latter one overhead starts to be prominent when compared to directly dividing the iterations between the threads with a static distribution.

- With respect to the results observed in the small test case, the maximum speedup is reached with 40 threads, with the OpenMP static scheduling option (13.83x) and 32 threads with the guided one (11.13x). After that number of threads, the efficiency slightly decreases due to the fact that the load per thread is not sufficiently large as to be able to leverage the instantiated resources. In the big test case, the highest speedup is observed with 64 threads with the static policy (17.81x) and 40 threads with the guided one (13.34x).
- Regarding the GPU-based performance, the comparison with the results observed when using CPU threads is clearly dependant on the mesh dimension, as in the 2D case. When computing the FTLE for small meshes, we can see in the graphic that, from 24 to 112 CPU threads, the performance with the static policy and a multithreaded CPU execution is better than that offered with only one GPU. When the mesh is big, one GPU is outperformed when using 24 or more threads (due to a higher computational load). In any case, taking advantage of two or more GPU devices always outperforms the results observed with CPU threads (except for the case of using 64 threads and the static policy in the big test case, that is slightly faster than using 2 GPUs).
- Regarding the GPU scalability, as reflected in Table 13, we observe that the efficiency is between 71% and 83%.

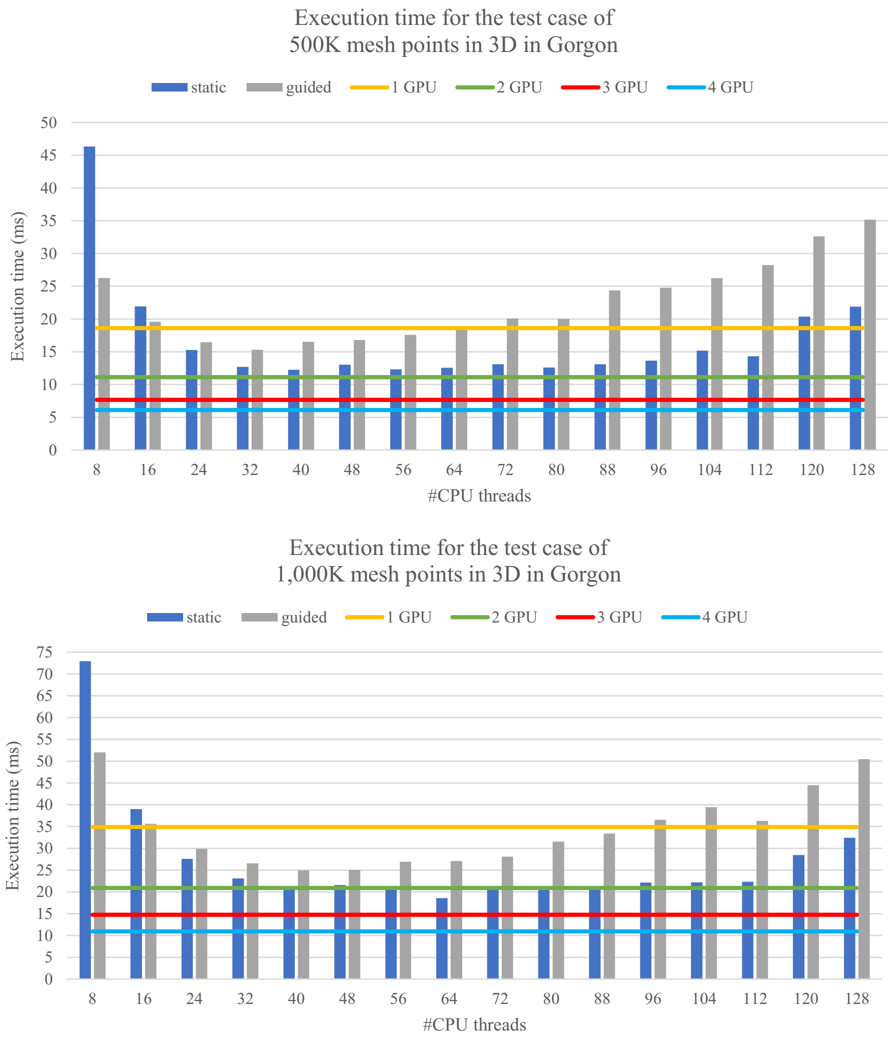

Fig. 3 Execution time for the 3D case with a mesh of 500K points (top) and 1000K points (bottom) in `Gorgon`. The GPU results are shown in horizontal lines, while the bars are used to represent the different execution time values associated to the CPU threads indicated in the X-axis

6 Concluding remarks

In this paper, we have detailed and analyzed the procedures to perform the FTLE computation. Moreover, we have provided an open source implementation of it, equipped with OpenMP directives to take advantage of multithreaded environments, as well as CUDA kernels to take advantage of any NVIDIA GPU devices.

In addition, we have offered an analysis of the performance attainable by our implementation, both when executed in multithreaded systems and also when using

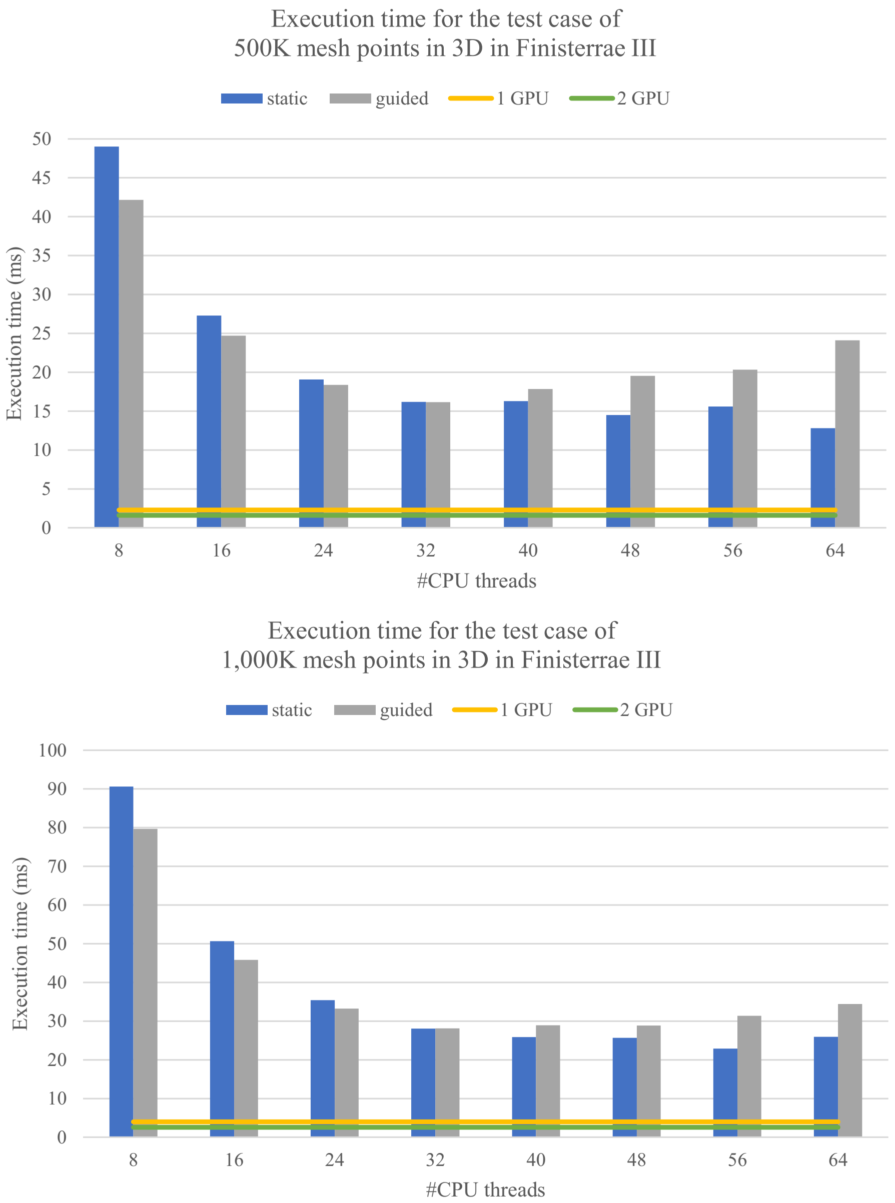

Fig. 4 Execution time for the 3D case with a mesh of 500K points (top) and 1000K points (bottom) in `Finisterrae III`. The GPU results are shown in horizontal lines, while the bars are used to represent the different execution time values associated to the CPU threads indicated in the X-axis

GPU devices (particularly, NVIDIA Titan Black and NVIDIA A100). On the one hand, this analysis has explored the impact of different OpenMP scheduling policies (`static`, `dynamic`, and `guided`); the conclusion being that the `static` policy

Table 13 GPU speedup and efficiency in `Gorgon` and `Finisterrae III` for the 3D small (left) and big (right) test cases

Platform	Small case-3D 500K points				Big case-3D 1000K points			
	Gorgon		Finisterrae III		Gorgon		Finisterrae III	
# GPU	Speedup	Efficiency	Speedup	Efficiency	Speedup	Efficiency	Speedup	Efficiency
2	1.6728	0.8364	1.4256	0.7128	1.6695	0.8347	1.5546	0.7773
3	2.4291	0.8097	–	–	2.3649	0.7883	–	–
4	3.047	0.7617	–	–	3.1906	0.7977	–	–

is the recommended option to ensure a good efficiency in any case. On the other hand, we have also compared that performance with the one attained when using up to four GPUs, concluding that multi-GPU executions outperform those based on CPU threads. Our experiments have covered both 2D and 3D FTLE computations, in two different architectures, reaching a notable efficiency in both scenarios, especially when taking advantage of the GPUs.

As stated before, our code is publicly available in GitHub, and Docker containers have been published in Docker Hub to ease the reproducibility and contribute to open science. We encourage the community to use our code and contact us for any inquiry.

7 Future work

As part of future work we plan to explore the performance of combining multi-CPU and multi-GPU parallelism. Moreover, we would also like to explore the use of FPGAs to exploit heterogeneous systems equipped with both GPUs and FPGAs.

Acknowledgements This work has been funded by the Consejería de Educación of Junta de Castilla y León, Ministerio de Economía, Industria y Competitividad of Spain, European Regional Development Fund (ERDF) program: Project PCAS (TIN2017-88614-R) and Project PROPHET-2 (VA226P20). This work was supported in part by grant TED2021-130367B-I00 funded by MCIN/AEI/10.13039/501100011033 and by "European Union NextGenerationEU/PRTR". Jose Sierra-Pallares was supported by project VA182P20 from Junta de Castilla y León. The experiments carried out using the CESGA resources were possible thanks to the Red Española de Supercomputación (RES) projects IM-2022-2-0015 and IM-2022-3-0021.

Author contributions RC-S, YT, SL-H, and JS-P implemented the code. RC-S, YT and SL-H carried out the experimentation and generated the figures. All authors wrote and reviewed the main manuscript text.

Funding Open Access funding provided thanks to the CRUE-CSIC agreement with Springer Nature. Consejería de Educación of Junta de Castilla y León, Ministerio de Economía, Industria y Competitividad of Spain, European Regional Development Fund (ERDF) programs: Project PCAS (TIN2017-88614-R), Project VA182P2, Project PROPHET-2 VA226P20. MCIN/AEI/10.13039/501100011033 and "European Union NextGenerationEU/PRTR" grant: TED2021-130367B-I00. Project IM-2022-2-0015 from the Red Española de Supercomputación (RES). Project IM-2022-3-0021 from the Red Española de Supercomputación (RES).

Data availability The source code is available at https://github.com/uva-trasgo/UVaFTLE/. Besides, we have created a Docker container that includes the FTLE source code (Docker Hub: https://hub.docker.com/r/rociocarratalasaez/uvaftle and Github Container Repository: https://github.com/uva-trasgo/UVaFTLE/pkgs/container/uvaftle) and a Docker container that allows the mesh and flowmap to be generated for use as input in the FTLE computation (Docker Hub: https://hub.docker.com/r/rociocarratalasaez/uvaftle-mesh-generation and Github Container Repository: https://github.com/uva-trasgo/UVaFTLE/pkgs/container/uvaftle-mesh-generation).

Declarations

Conflict of interest The authors have no competing interests as defined by Springer, or other interests that might be perceived to influence the results and/or discussion reported in this paper.

References

1. Abinit (2021) flowtk Package. https://abinit.github.io/abipy/api/flowtk_api.html, accessed: Dec 2022
2. Barroso LA, Clidaras J, Hölzle U (2013) The datacenter as a computer: an introduction to the design of warehouse-scale machines, Second Edition. Morgan & Claypool Publishers, http://dx.doi.org/10.2200/S00516ED2V01Y201306CAC024
3. Bedford Taylor M (2017) The evolution of bitcoin hardware. Computer 50(9):58–66. https://doi.org/10.1109/MC.2017.3571056
4. Betz J, Zheng H, Liniger A et al (2022) Autonomous vehicles on the edge: a survey on autonomous vehicle racing. IEEE Open J Intell Transp Syst 3:458–488. https://doi.org/10.1109/OJITS.2022.3181510
5. Brodtkorb AR, Dyken C, Hagen TR et al (2010) State-of-the-art in heterogeneous computing. Sci Program. https://doi.org/10.1155/2010/540159
6. Brunton S, Rowley C (2009) Modeling the unsteady aerodynamic forces on small-scale wings. In: 47th AIAA aerospace sciences meeting including the new horizons forum and aerospace exposition, p 1127, https://doi.org/10.2514/6.2009-1127
7. Carratalá-Sáez R, Sierra-Pallares J, Llanos DR et al (2022) UVaFlow: Lagrangian flowmap computation for fluid dynamic applications. Submitted to the Journal of Computational Science
8. Chen CM, Shen HW (2013) Graph-based seed scheduling for out-of-core ftle and pathline computation. In: 2013 IEEE Symposium on Large-Scale Data Analysis and Visualization (LDAV), pp 15–23, https://doi.org/10.1109/LDAV.2013.6675154
9. Childs H, Brugger E, Whitlock B et al (2012) Visit: an end-user tool for visualizing and analyzing very large data. In: High performance visualization–enabling extreme-scale scientific insight. Taylor & Francis, p 357–372, https://doi.org/10.1201/b12985
10. Conti C, Rossinelli D, Koumoutsakos P (2012) GPU and APU computations of finite time Lyapunov exponent fields. J Comput Phys 231(5):2229–2244. https://doi.org/10.1016/j.jcp.2011.10.032
11. Coulliette C, Wiggins S (2000) Intergyre transport in a wind-driven, quasigeostrophic double gyre: an application of lobe dynamics. Nonlinear Process Geophys 7(1/2):59–85. https://doi.org/10.5194/npg-7-59-2000
12. Dauch T, Rapp T, Chaussonnet G et al (2018) Highly efficient computation of finite-time Lyapunov exponents (FTLE) on GPUs based on three-dimensional SPH datasets. Comput Fluids 175:129–141

13. Garth C, Gerhardt F, Tricoche X et al (2007) Efficient computation and visualization of coherent structures in fluid flow applications. IEEE Trans Visual Comput Graphics 13(6):1464–1471. https://doi.org/10.1109/TVCG.2007.70551
14. Garth C, Li GS, Tricoche X et al (2009) Visualization of coherent structures in transient 2D flows, Springer: Berlin, Heidelberg, pp 1–13. https://doi.org/10.1007/978-3-540-88606-8_1
15. Haller G (2015) Lagrangian coherent structures. Annu Rev Fluid Mech 47:137–162. https://doi.org/10.1063/1.3690153
16. Hlawatsch M, Sadlo F, Weiskopf D (2011) Hierarchical line integration. IEEE Trans Visual Comput Gr 17(8):1148–1163. https://doi.org/10.1109/TVCG.2010.227
17. Klöckner A, Pinto N, Lee Y et al (2012) PyCUDA and PyOpenCL: a scripting-based approach to GPU run-time code generation. Parallel Comput 38(3):157–174. https://doi.org/10.1016/j.parco.2011.09.001
18. Kuhn A, Rössl C, Weinkauf T et al (2012) A benchmark for evaluating ftle computations. In: 2012 IEEE Pacific visualization symposium, pp 121–128, https://doi.org/10.1109/PacificVis.2012.6183582
19. Lin M, Xu M, Fu X (2017) GPU-accelerated computing for Lagrangian coherent structures of multi-body gravitational regimes. Astrophys Space Sci 362:1572–946X. https://doi.org/10.1007/s10509-017-3050-y
20. Mavriplis DJ (1997) Unstructured grid techniques. Annu Rev Fluid Mech 29(1):473–514
21. Meschi SS, Farghadan A, Arzani A (2021) Flow topology and targeted drug delivery in cardiovascular disease. J Biomech 119(110):307. https://doi.org/10.1016/j.jbiomech.2021.110307
22. Mikolajczak M (1997) Designing and building parallel programs: concepts and tools for parallel software engineering [book review]. IEEE Concurr 5(2):88–90. https://doi.org/10.1109/MCC.1997.588301
23. Nouanesengsy B, Lee TY, Lu K et al (2012) Parallel Particle Advection and FTLE Computation for Time-Varying Flow Fields. In: SC '12: Proceedings of the International Conference on High Performance Computing, Networking, Storage and Analysis, pp 1–11, https://doi.org/10.1109/SC.2012.93
24. NVIDIA (2022a) CUDA C++ Programming Guide. On https://docs.nvidia.com/cuda/pdf/CUDA_C_Programming_Guide.pdf
25. NVIDIA (2022b) CUDA Profiler Guide. On https://docs.nvidia.com/cuda/pdf/CUDA_Profiler_Users_Guide.pdf
26. Onu K, Huhn F, Haller G (2015) Lcs tool: a computational platform for Lagrangian coherent structures. J Comput Sci 7:26–36. https://doi.org/10.1016/j.jocs.2014.12.002
27. Sadlo F, Rigazzi A, Peikert R (2011) Time-Dependent Visualization of Lagrangian Coherent Structures by Grid Advection, Springer: Berlin, Heidelberg, pp 151–165. https://doi.org/10.1007/978-3-642-15014-2_13
28. Sagristà A, Jordan S, Sadlo F (2020) Visual analysis of the finite-time Lyapunov exponent. Comput Graph Forum 39(3):331–342. https://doi.org/10.1111/cgf.13984
29. Serra M, Sathe P, Beron-Vera F et al (2017) Uncovering the edge of the polar vortex. J Atmos Sci 74(11):3871–3885. https://doi.org/10.1175/JAS-D-17-0052.1
30. Spaulding ML (2017) State of the art review and future directions in oil spill modeling. Mar Pollut Bull 115(1–2):7–19. https://doi.org/10.1016/j.marpolbul.2017.01.001
31. TOP500.org (2022) Top500 Supercomput. Sites. On http://www.top500.org
32. Wang F, Deng L, Zhao D et al (2016) An Efficient Preprocessing and Composition Based Finite-Time Lyapunov Exponent Visualization Algorithm for Unsteady Flow Field. In: 2016 International Conference on Virtual Reality and Visualization (ICVRV), pp 497–502, https://doi.org/10.1109/ICVRV.2016.89
33. Xuan H, Wei S, Li Y et al (2019) Off-line time aware scheduling of bag-of-tasks on heterogeneous distributed system. IEEE Access. https://doi.org/10.1109/ACCESS.2019.2899926
34. Zahran M (2019) Heterogeneous computing: hardware and software perspectives, vol 23. Association for computing machinery, New York, NY, USA. https://doi.org/10.1145/3281649
35. Zhao XH, Kwek KH, Li JB et al (1993) Chaotic and resonant streamlines in the ABC flow. SIAM J Appl Math 53(1):71–77

Authors and Affiliations

Rocío Carratalá-Sáez[1] · Yuri Torres[1] · José Sierra-Pallares[2] · Sergio López-Huguet[3] · Diego R. Llanos[1]

Yuri Torres
yuri.torres@infor.uva.es

José Sierra-Pallares
jsierra@eii.uva.es

Sergio López-Huguet
serlohu@upv.es

Diego R. Llanos
diego@infor.uva.es

[1] Depto. Informática, Universidad de Valladolid, Paseo de Belén, 15, Valladolid 47011, Castilla y León, Spain

[2] Depto. Ingeniería Energética y Fluidomecánica, Universidad de Valladolid, Paseo del Cauce, 59, Valladolid 47011, Castilla y León, Spain

[3] Instituto de Instrumentación para Imagen Molecular (I3M), Universitat Politècnica de València, Camino de Vera S/N, València 46022, Comunidad Valenciana, Spain